uc3m | Universidad Carlos III de Madrid

Data Structures and Algorithms.
Author: Isabel Segura Bedmar

Unit 1 – Abstract Data Types

Problem - Our first Python class, CreditCard.

Design and implement a class to represent the credit cards for bank customers. Every credit card must have the following data:

- the name of the customer.
- an id for the credit card.
- the balance of the credit card, which is the money spent.
- the limit of the credit card, which is the total amount that the customer can spend.

Moreover, the following operations must be considered:

- charge, which takes an amount of money and increases the balance of the credit card with this amount. If the new balance (current balance plus charge) exceeds the limit of the credit card, the operation must no be performed.
- make_deposit, which takes an amount of money and decreases the balance. The deposit should not exceed the current balance.

Note: Please, stick to the specifications mentioned above, do not make different assumptions about how credit cards should work. This is only an exercise to practice programming classes with Python.

Solution:

```
class CreditCard:

  """The CreditCard class provide a simple model, a template for credit
cards.

A credit card must contain information about the customer, account number,
credit limit, and

current balance."""

  def __init__ (self, name, idCard, limit):

    """Creates a new credit card object"""

    self._customer=name #the name of the customer

    self._idCard=idCard #the id of the credit card

    self._limit=limit   #credit limit

    self._balance=0     #the initial balance is 0



  def charge(self, price):

    """Charge the price to the balance of the card.

    Must check if there is enough credit. Return True

    if the charge was processed, False if charge was denied"""

    # if charge would exceed limit,

    if price + self._balance > self._limit:

      print('charge denied')

    else:

      self._balance = self._balance + price


  def make_deposit(self, amount):

    """Make a deposit (add money to the credit card).

    Reducs the balance"""

    self._balance = self._balance - amount

    if self._balance<0:

      print('Balance cannot be negative')

      self._balance=0
```

```
 def show(self):
    print('Credit card information:')
    print('Customer:%s'%self._customer)
    print('Id:%s'%self._idCard)
    print('Limit:%d'%self._limit)
    print('Balance:%d'%self._balance)
    print('--------------------------')

"""Now, we create an instance of a credit card and try each method."""

#Create an instance of the credit card
cc1 = CreditCard('Isabel Segura','5391 0375 9387 5309', 3000 )
cc1.show()

#we make a charge
cc1.charge(2000)
cc1.show()

#we make a deposit
cc1.make_deposit(1000)
cc1.show()

cc1.charge(2500)
cc1.show()
```

Problem - Multidimensional Vector Class

Please, implement a class, **Vector**, to represent the coordinates of a vector in a multidimensional space. For example:

In a three-dimensional space, we might wish to represent a vector with the following coordinates: 5, −2, 3 .

In a five-dimensional space, a possible vector may have the following coordinates: 0,1,-1,3,2.

The class must contain the following methods:

- **__init__**(**self**,dim): constructor methods that creates a vector of dimension dim. In this method,, all coordinates of the vector are equal to 0.
- **__len__**(**self**): returns the dimension of the vector.
- **__str__**(**self**): returns a string that represents the vector. For example, if the coordinates of the vector are: 3,5,0, the method should return the string "(3,5,0)".
- getItem(**self**,i): returns the ith coordinate of the vector. The first coordinate is always represented by the index 0.
- setItem(**self**,i,newValue): modifies the ith coordinate of the vector to the given newValue.
- __eq__(**self**,other): returns True if the invoking vector and the other vector are equal, and false otherwise
- sumVector(**self**,other): returns a new vector, which is the sum of the invoking vector and the param other.

Solution:

```
class Vector:

  def __init__(self,dim):
    """Creates a vector of dimension dim.
    All its coordinates are 0's"""
    self.items=[0]*dim

  def __len__(self):
    """Returns the dimension of the vector."""
    return len(self.items)

  def getItem(self,i):
    """Returns the ith coordinate of the vector"""
    return self.items[i]

  def setItem(self,i,newValue):
    """Sets the ith coordinate to the given value"""
    if i<0 or i>=len(self):
```

```
      print('Error: index out of range')
      return

    self.items[i]=newValue

  def __str__(self):
    """Returns a string containing the vector"""
    result='('
    for i in range(0,len(self)):
      result  = result + str(self.getItem(i)) + ','
    result=result[:-1]
    result+=')'
    return result


  def __eq__(self,other):
    """checks if the two vectors have the same coordinates"""
    if len(self)!=len(other):
      return False

    for i in range(0,len(self)):
      if self.getItem(i) != other.getItem(i):
        return False

    return True


def sumVector(self,other):
     """Returns a new vector, which is the sum of the invoking vector and the
   param other"""

    if len(self)!=len(other):
      print('Error: vectors with different dimensions')
      return None

    #creates a new vector
    sumV=Vector(dim)
    for i in range(0,len(self)):
      sumV.setItem(i,self.getItem(i)+other.getItem(i))

    return sumV

"""Now, we test the class and its methods."""

import random
dim=4
v1=Vector(dim)
v2=Vector(dim)
v3=Vector(dim)

#This loop allows us to initialize the coordinates of the two vectors
```

```
#in a random way, with values from 0 to 99
for i in range(0,dim):
  v1.setItem(i,random.randint(0,100))
  v2.setItem(i,random.randint(0,100))
  v3.setItem(i,v1.getItem(i))

#we show the three vectors vectors
print("v1={}".format(str(v1)))
print("v2={}".format(str(v2)))
print("v3={}".format(str(v3)))


#we check if they are equal
print("{}=={}?={}".format(v1,v3,v1==v3))
print("{}=={}?={}".format(v1,v2,v1==v2))


#we sum the two vectors

print("{}+{}={}".format(v1,v2,v1.sumVector(v2)))
```

Problem - Date ADT

Implement a class, Date, to represent dates. A date represents a single day in our calendar (e.g December 25, 2018 AC).The operations are:

- **__init__**(*day,month,year*): creates a new date instance.
- **_str__**(): returns a string representation in the format '*dd/mm'/yyyy*'.
- **day**(): returns the day number of this date.
- **month**(): returns the month number of this date.
- **year**(): returns the year number of this date.
- **monthName**(): returns the month name of this date.
- **isLeapYear**(): return True if this date falls in a leap year, and False otherwise.
- **compareTo**(other): compares this date to the *other* to determine their logical order.
    - If this date is before *other*, returns -1.
    - If both dates are the same, return 0. I
    - If this date is after *other*, returns 1.

Solution:

```
MONTH={1:'January',2:'February',3:'March',4:'April',5:'May',
      6:'June', 7:'July', 8:'August', 9:'September',
      10:'October',11:'November',12:'December'}
```

```
class Date:


  def __init__(self, day, month, year):
    self.day = day
    self.month = month
    self.year = year

  def __str__(self):
    return str(self.day)+'/'+str(self.month)+'/'+str(self.year)

  def day(self):
    return self.day

  def month(self):
    return self.month

  def year(self):
    return self.year

  def monthName(self):
    return MONTH[self.month]



  def isLeapYear(self):
    """Determine whether a year is a leap year."""
    return self.year % 4 == 0 and (self.year % 100 != 0 or self.year % 400 ==
0)

  def compareTo(self,other):
    if self.year<other.year:
      return -1

    if self.year>other.year:
      return 1

    #if this line is reached, it means that both years are equal
    if self.month<other.month:
      return -1

    if self.month>other.month:
      return 1

    #if this line is reached, it means that both months are equal
    if self.day<other.day:
      return -1

    if self.day>other.day:
      return 1

    #if this line is reached, it means that both days are equal
    return 0

"""Now, we create some instances (objects) of the Date class, and test some
methods."""

d1=Date(3,10,2016)
```

```
print("Date: {} ".format(str(d1)))
print("Its month name is {}".format(d1.monthName()))
print("Is it a leap year? {}".format(d1.isLeapYear()))


def compareDates(d1,d2):
  if d1.compareTo(d2) == -1:
    print("{} is lower than {} ".format(str(d1),str(d2)))
  elif d1.compareTo(d2) == 1:
    print("{} is greater than {} ".format(str(d1),str(d2)))
  else:
    print("{} and {} are equal ".format(str(d1),str(d2)))

d2=Date(3,10,2018)
compareDates(d1,d2)

d3=Date(1,1,2016)
compareDates(d1,d3)

d4=Date(3,10,2016)
compareDates(d1,d4)

"""## Using the Python's date class

Actually, you don't need to implement the Date class, because Python already
includes a class for representing dates, the datetime class in the package
datetime. The following cell shows how you can work with this class:
"""

import datetime

# date in yyyy/mm/dd format
d1 = datetime.datetime(2018, 5, 3)
d2 = datetime.datetime(2018, 6, 1)

# Comparing the dates will return
# either True or False
print("d1 is greater than d2 : ", d1 > d2)
print("d1 is less than d2 : ", d1 < d2)
print("d1 is not equal to d2 : ", d1 != d2)

print( d1.year, d1.month, d1.day)
#where 0 denotes Monday.
print(d1.weekday())
```

Problem – The Polynomial ADT

A polynomial is an expression representing a mathematical sum of several terms. Each term has a number called the coefficient, a variable and a power of the variable called the exponent.

$Q(x)=a_0+a_1x+a_2x^2+a_3x^3+\ldots+a_nx^n$

For example, a_1 is the coefficient for term of degree 1, a_2 is the coefficient for term of degree 3, and so on. a_0 is the term of degree 0, is also named as constant term of

the polynomial.

The operations for the polynomial ADT are:

- **__init__**(coef): creates a new polynomial whose coefficients are the elements of the input list coef. The element at index 0 is the coefficient of term with degree 0 (constant term), the element at index 1 is the coefficient of term with degree 1, and so on.
- **getDegree**(), which returns the polynomial grade. For example, Q(x)=5 has degree 0. $Q(x)=x^2+5$ has degree 2.
- **getCoefficient**(n), which returns the coefficient of the term, which is squared to n. For example, given the polynomial $Q(x)=a_0+a_1x+a_2x^2+a_3x^3+\ldots+a_nx^n$, this call getCoefficient(3) returns a_3.
- **setCoefficient**(n, newValue), which modifies the coefficient of the term whose power is n by the value newValue. For example, given the polynomial $Q(x)=a_0+a_1x+a_2x^2+a_3x^3+\ldots+a_nx^n$, this call setCoefficient(3,b) does that now Q(x) is $a_0+a_1x+a_2x^2+bx^3+\ldots+a_nx^n$
- **evaluate**(x), which takes x as param and returns the value of the polynomial functions for this value. For example, $Q(3)=a_0+a_13+a_29+a_327+\ldots+a_n3^n$
- **sum**(p), which returns the sum of the invoking polynomial and the polynomial p. The invoking polynomial must not be modified. For example, $Q(x)=3x^2+4x+5$, $p(x)=x^3+4x^2-2x-3$. Q.sum(p) -> x^3+7x^2+2x+2 .

Note: Use a Python list to store the polynomial coefficients. What is the best way to store the coefficients into the list?. In what position of the list is it better to store the constant term?. Why?

Solution:

```
class Polynomial:
  """Python class to represent polynomial functions"""

  def __init__(self, coefficients):
       """This constructor takes the coefficients for the terms of the
polynomial.
     We assume that the constant term (degree 0) is stored at the index 0 in
the list,
    the term with degree 1 is at the index 1, and so on..."""
    self.coefficients = coefficients

  def degree(self):
    """It returns the degree of the polynomial"""
    return len(self.coefficients)-1

  def getCoefficient(self,i):
    """It returns the coefficient of the term with degree i"""
```

```
    if i<0 or i>self.degree():
      print('{} index out of range'.format(i))
      return None

    #if i not in range(0,self.degree()+1):
    #  print('{} index out of range'.format(i))
    #  return None

    return self.coefficients[i]


  def setCoefficient(self,i,newValue):
    """It modifies the coefficient of the term with degree i to newValue"""


    if i not in range(0,self.degree()+1):
      print('{} index out of range'.format(i))
      return

    self.coefficients[i]=newValue


  def evaluate(self,x):
    "This method returns the value of the polynomial functions for x"
    result=0
    for i in range(0,self.degree()+1):
      result += self.getCoefficient(i)*pow(x,i)

    return result



  def sum(self,q):
       """It returns a new polynomial which is the sum of the invoking
polynomial (self)
    and q. """

     #Create a new polynomial that is a copy of the polynomial with greater
degree
    if (self.degree()>=q.degree()):
      sumPol=Polynomial(self.coefficients)
      #now, we have to add the coefficients from q
      for i in range(0,q.degree()+1):
                                                  sumPol.setCoefficient(i,
sumPol.getCoefficient(i)+q.getCoefficient(i))
    else:
      sumPol=Polynomial(q.coefficients)
      #now, we have to add the coefficients from self
      for i in range(0,self.degree()+1):
                                                  sumPol.setCoefficient(i,
sumPol.getCoefficient(i)+self.getCoefficient(i))
```

```
    return sumPol


  def toString(self):
    "It retuns a string representing the polynomial function"
    constant=self.getCoefficient(0)

    if constant!=0:
      result=str(constant)
    else:
      result=''

    for i in range(1,self.degree()+1):
      term=self.getCoefficient(i)
      if term!=0:
        if result!='' and term>0:
          result=result+str('+')
        if term==1:
          term=''
        elif term==-1:
          term='-'
        result  = result + str(term)+str('x')

        if i>1:
          result=result+str('^')+str(i)

    return result

"""Now, we include some instructions for testing the different methods"""

#we create a polynomial
p=Polynomial([1,2,3,0,0,-2,8])

#we test the method toString
print('p={}'.format(p.toString()))

#we test the method degree
print('Degree:{}'.format(p.degree()))

#we test the method getCoefficient for different indexes
print('getCoefficient(0)={}'.format(p.getCoefficient(0)))
print('getCoefficient(1)={}'.format(p.getCoefficient(1)))
print('getCoefficient(2)={}'.format(p.getCoefficient(2)))
print('getCoefficient(3)={}'.format(p.getCoefficient(3)))


#we test the method setCoefficient for several values
p.setCoefficient(0,0)
print('setCoefficient(0,0)={}'.format(p.toString()))
```

```
p.setCoefficient(0,5)
print('setCoefficient(0,5)={}'.format(p.toString()))

p.setCoefficient(1,-1)
print('setCoefficient(1,-1)={}'.format(p.toString()))

p.setCoefficient(2,1)
print('setCoefficient(2,1)={}'.format(p.toString()))

p.setCoefficient(3,4)
print('setCoefficient(3,4)={}'.format(p.toString()))

#we test the method evaluate for several values
print('evaluate(0)={}'.format(p.evaluate(0)))
print('evaluate(1)={}'.format(p.evaluate(1)))
print('evaluate(2)={}'.format(p.evaluate(2)))


q=Polynomial([3,-3,7,2,0,1])
print('q={}'.format(q.toString()))
print('p+q={}'.format(p.sum(q).toString()))
```